

ROAR End-to-End Reinforcement Learning

Xuan Liu, Federico Palacios, and Franco Leonardo Huang

University of California Berkeley

Author Note

All authors contributed equally to this research. Correspondence concerning this article should be addressed to either of authors, E-mail: lx971223@berkeley.edu, franco_huang@berkeley.edu, fedepal45@berkeley.edu

Abstract

Serious crashes are usually due to human errors and self-driving would significantly reduce the number of accidents. However, training a car to drive itself requires failures, but crashing in the real world is too expensive and dangerous. Therefore, in our project, we train a car to race around a virtual map of Berkeley and proposed to use end-to-end reinforcement learning algorithms to help us explore as many states as possible and learn how to drive in that situation. Inside the car agent, we adopt state-of-the-art Reinforcement Learning algorithms such as Proximal Policy Optimization to teach our model to navigate the course. With an End-to-end implementation, we can generalize our model to more situations than previous proportional–integral–derivative (PID) controller versions can.

Keywords: ROAR competition, Self-driving, Reinforcement Learning, PPO agent, Carla environment

ROAR-RL	3
---------	---

Summary

Introduction	4
Related Work	6
STRUCTURE AND TECHNIQUE	9
Carla Environment	10
Observation Agent	11
Feature Extraction Model	15
Proximal Policy Optimization Algorithm	15
Calculation of Action and Reward	17
Evaluation	17
Steering only control	17
Full Control	29
Berkeley Major	31
Conclusion	32
Future Work	33

ROAR End-to-End Reinforcement Learning

Introduction

Autonomous driving provides a wide range of benefits. First and foremost is safety. According to the National Highway Traffic Safety Administration, 94% of serious crashes are due to human error (NHTSA, 2021). Automation would significantly reduce the number of accidents and has the potential to save thousands of lives every year. Another benefit to automated driving is the reduction of traffic congestion. With numerous vehicles becoming autonomous, it becomes easier to control traffic flow. Autonomous cars also provide independence to individuals who cannot operate classic cars due to disability.

Although autonomous driving has many advantages, there are still many challenges and problems in its application. One problem is that the expense of testing has risen dramatically when developing new algorithms for autopilot. Small to midsize businesses and major research colleges cannot afford the computer expenditures of autonomous driving and managing an extensive fleet of vehicles. In addition, the cost of crashing and ruining the self-driving car during the test is exorbitant. The second problem is that driving is particularly challenging for artificial intelligence. One explanation is that artificial intelligence is not as adaptive as the human brain, which can fail in unexpected ways. Although self-driving cars may have a good understanding of what other vehicles are capable of, pedestrians are unpredictable and can alter their attitudes at any time. Engineers now have to be able to manage and deploy a system that can accommodate 99 percent predictable material and a small amount of uncertain content. The final 1% is exceptionally challenging, which is why self-driving cars are currently off the road.

Fortunately, ROAR addresses this issue by creating low-cost artificial intelligence, autonomous driving software, and open-source hardware reference designs. ROAR's hardware design costs less than \$500 and can construct a self-driving remote-controlled automobile that can reach speeds of more than 70 miles per hour. Meanwhile, the ROAR competition encourages the public to solve the second issue by providing a platform that

includes a simulated environment in which competitors can test their self-driving algorithms. Our goal is to develop an agent as accurately as possible for self-driving in such a simulation environment. Traditionally, manually tuned PID-style controllers dominated the ROAR competition. While they have been effective, this method is not generalizable and has limited exploration for a truly optimal solution. As a result, we are training our model in the game setting of car racing. The simulator contains the roads and buildings for UCB, and the goal of the car racing is to complete a lap around it. There are four sharp turns for the Berkeley minor map and ten sharp turns for those in the Berkeley major in the lap. The player can only see the nearby view during the drive and control the throttle, steering, and brake. To simplify the situation, there is only one car racing at a time in the simulator. Also, only the racing time is considered.

We propose to use end-to-end reinforcement learning algorithms to help us explore as many states as possible and learn how to drive in that situation. Specifically, we use the occupancy grid map of each frame as input to allow the agent to automatically sense the current condition, as opposed to the prior PID method, which only considers waypoints to enable vehicles to pass through the pre-set racing line as rapidly as feasible. On the other hand, the occupancy grid map is more concise and more accessible to comprehend by models than the real-world image data. Before we passed the input to the encoder, we extracted the last four frames for each frame and did frame stacking to incorporate more information. Then our four stacked input frames go through a Convolutional Neural Network (CNN) model, flattened and passed into several fully connected layers. These CNN layers help the model understand what the changes in time represented in the input data. We then pass the features to our Proximal Policy Optimization or PPO algorithm. This On-Policy algorithm will take our features and reward information to decide what course of action to take next. Finally, PPO passes its determined outputs to CARLA(Car Learning to Act) for execution.

Related Work

Reinforcement learning has a wide range of applications in autonomous driving. For example, it can be applied to a part of self-driving like planning and optimizing driving trajectories or directly acting as an end-to-end agent that collects input from the environment and directly outputs the car's actions. Figure 1 shows the interaction of reinforcement learning.

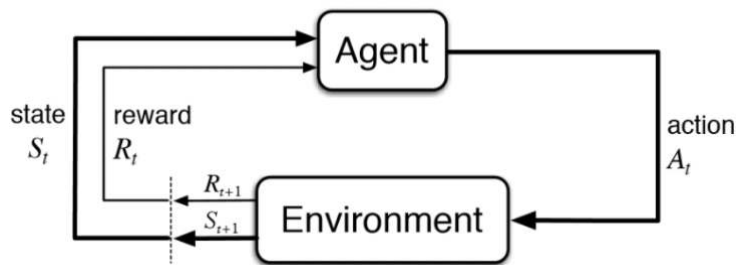


Figure 1

What is Reinforcement Learning

"Safe Trajectory Planning Using Reinforcement Learning for Self Driving" was trying to use a reinforcement learning algorithm to help their agent find a safe trajectory in self-driving.(Coad et al., 2020) This is a typical non-end-to-end reinforcement learning application. Because the end-to-end learning process is more challenging, only the agent is responsible for planning the lane. Then the lane following procedure is delegated to other controllers. Some researchers (Mania et al., 2018) also compare the classic optimal control method with the reinforcement learning method to find their performance difference in trajectory planning.

Early word end-to-end reinforcement learning application, using behavior cloning to learn from demonstration, which means to imitate the behavior of experts as much as possible(Pomerleau, 1989). Later, "Learning driving styles for autonomous vehicles from demonstration" recommends an expert demonstration of a human driver using the maximum entropy inverse RL to learn comfortable driving trajectory

optimization.(Kuderer et al., 2015) This work was done by behavior cloning algorithms, a supervised learning reinforcement learning algorithm. Although behavior cloning can be applied to self-driving cars, this method has obvious drawbacks. It is difficult for the agent to adapt to brand new situations. In addition, there will be a problem of prediction bias when a minor change happens to the environment.

After that, researchers began to use a DQN-like reinforcement learning algorithm to assist self-driving(Sharifzadeh et al., 2017) (Li & Czarnecki, 2019). In addition, there exist some model-based deep RL algorithms which are directly processing pixel inputs and learn policies(Wahlström et al., 2014). When it comes to real-world settings, "Learning to Drive in a Day" conducted simulation training and then used the onboard computer for real-time training(Kendall et al., 2018), the agent they trained was able to learn how to follow the lane and completed a real-world test on a 250-meter section.

When applying reinforcement learning to autonomous driving, the most important thing is how to define state spaces, action spaces, and rewards. A survey gives us a review of different state and action representations used in self-driving research (Leurent, 2018). Commonly used state-space features of autonomous vehicles include the position, orientation, and speed of the autonomous vehicle and the border and obstacles in the environment.

That said, the car's environment is taken out quite complex. We took inspiration from the Atari video game console to simplify it into a more manageable state. "A Graphic Guide to Implementing PPO for Atari Games" trained a Reinforcement Learning model to play the Atari game "Breakout" by observing the simple two-dimensional images rendered on the screen(**Atari**). It successfully determined what direction to move the paddle to achieve the highest score. In addition, it was able to give the model an understanding of motion bypassing four-screen frames at different moments in time. At first glance, the Atari game may seem very different from ours, but we can see a similar two-dimensional structure when focusing on the occupancy grid map.

Our model uses the occupancy grid map as the primary state representation, integrating vehicle position, orientations, obstacles, and manually set reward lines. For the design of actions, the continuous value actuators generally set in the industry for vehicle control include steering, throttle, and brake. Last but not least, the reward design has always been a point of value debate. Some examples include the distance to the destination, the speed of the self-vehicle, the collision of objects, avoiding extreme braking or steering, etc. (Kardell & Kuosku, 2017). We considered a number of the factors mentioned above in the model’s design and innovatively designed way lines perpendicular to the current heading as indicator to help the reinforcement learning agent accelerate the learning progress.

STRUCTURE AND TECHNIQUE

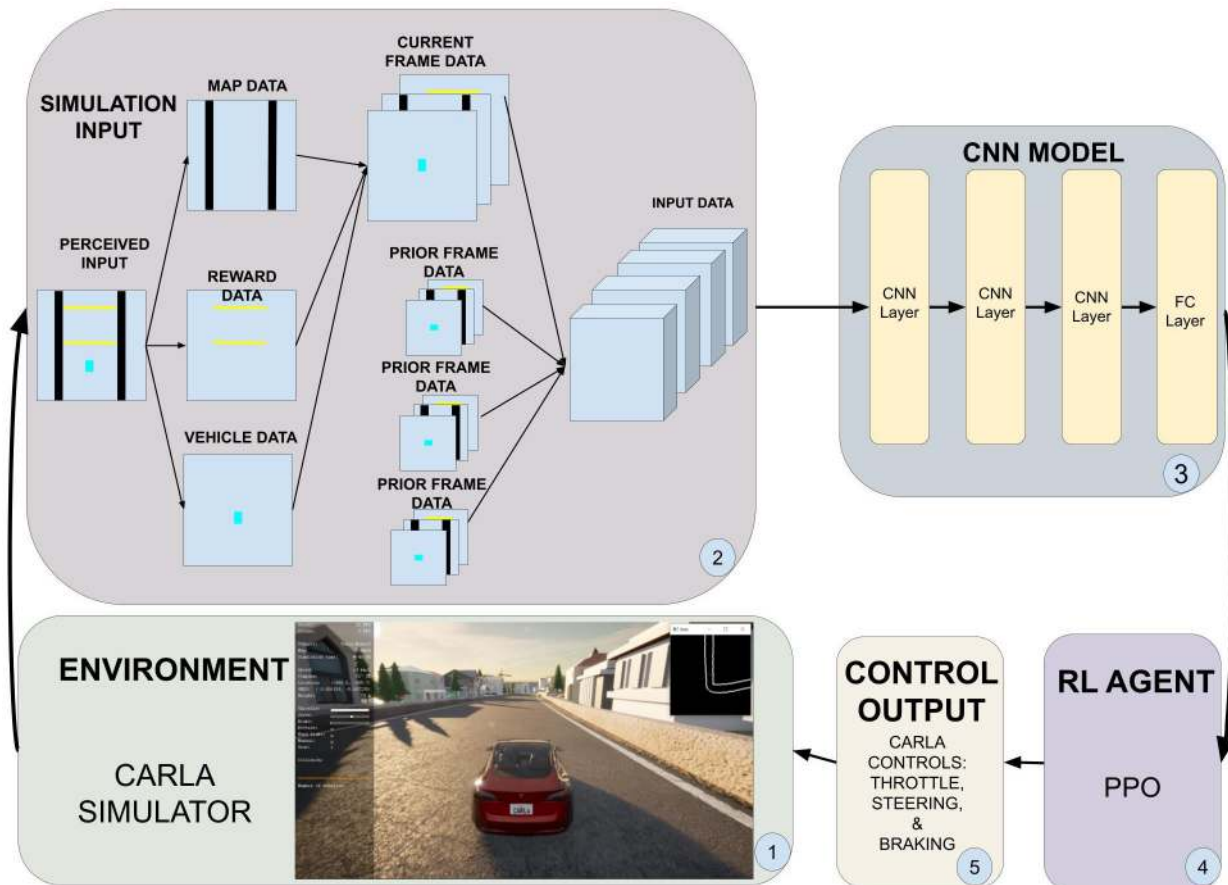


Figure 2

Representation of the Input Data

Our project consists of four parts, observation agent, feature extraction model, PPO agent, and environment. As the image 2 shows, during each run, to make an action decision based on the current state, the agent first generates the helpful information of the four last frames, including the occupancy grid map, reward lines, and vehicle state. Then, the feature extraction model receives the current state as the input to compute the features and feed them into the PPO agent. After that, the PPO agent calculates action based on the input and passes the action for the vehicle, throttle, steering, and braking, to the environment. Finally, the environment part uses the action to update the Carla simulator and save the current state to the agent.

Carla Environment

Our project is run in the CARLA Open-source simulator for autonomous driving research. Carla, also known as car learning to act, is an open-source simulator for autonomous driving research. The open-source simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors, map generation, etc.

Berkeley map. We use CARLA to recreate virtual models of our two racing tracks, "Berkeley Minor" and "Berkeley Major" (Both based on the UC Berkeley Campus). Berkeley Minor map contains four roads around UCB which form a lap. Berkeley Major map is an updated version of the Minor map, which includes a 10-times large lap and many more and sharper turns. The CARLA API also comes with various vehicles for us to insert and control. Several parameters, such as frame shape, acceleration, and wheel friction, are tuned to represent the vehicle model accurately. In our project, our agent uses a Tesla Model 3 to navigate the track.

Multiple Spawn Points. The spawn points of the car were pre-set by the CARLA environment. In the Berkeley minor map, there are only two spawn points, and the locations of both points are very close. And so, we were able to successfully train a model using a single spawn location for the "Berkeley Minor" track. However, when moving onto the much larger "Berkeley Major" we ran the risk of over-fitting to a particular section of the track before reaching new obstacles or types of turns. For example, if we spawned the model during a section of the course that had straight lines and easy, right-hand turns, it would learn to navigate that environment well. As the model progresses further and further down the path, this behavior will be reinforced and solidified. By the time it reaches the end of the course, where it is presented with an entirely new kind of obstacle, such as a sharp series of left turns, the model would struggle to adapt its known methods to this new problem.

A solution presented in this Youtube video describes spawning a vehicle in different

locations along with the map during training. This method exposes the model to several different scenarios and obstacles, which leads to a more generalized model. For Berkeley Major, we apply this method across twelve different spawn points, as shown in 3, as the map is very diverse. Once we have a general working model, we can begin to train from the initial spawn point of the race and focus on over-fitting to achieve an optimal racing line and the fastest time.

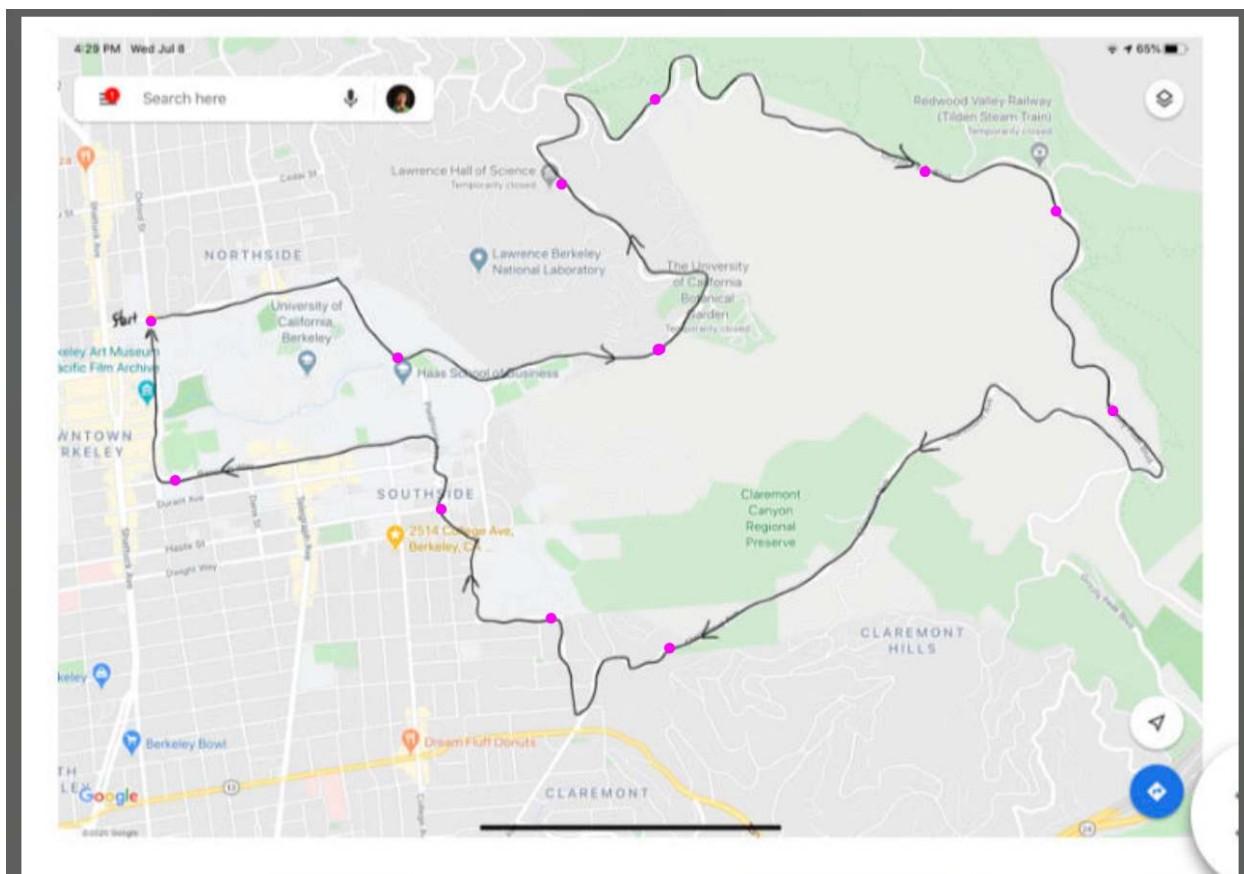


Figure 3

Multiple Spawn Points in Berkeley Major Map

Observation Agent

The OGM is a 2D array representing a bird's eye view of the map. The bird's eye view of the map has a more detailed and further exploration of the map. The map is first generated by manually driving the vehicle around the course and using the built-in sensors

(simulated by CARLA) to estimate the track and surroundings. After cleaning up the image, the clearly defined borders outline our course.

There are three components in the observation, occupancy grid map, reward lines, and vehicle state. The Occupancy Grid Map or OGM contains information about the simulation environment. Since our task is driving on the road, the wall locations are extracted as obstacles to being in OGM. The reward lines represent the step-by-step subtask to give a reward when it passes certain lines, which cut the whole loop into hundreds of segments. The vehicle state includes the car location and car rotation. Since the observation is ego-centric, the vehicle is always in a fixed place of the OGM, and the OGM is rotated so that the car is always facing upwards. The last four observations are stacked to show the movement and used for the model’s input. The figure below shows the result observation. Hence the input space is a matrix of dimension $4 \times 4 \times 84 \times 84$, in which 84×84 represents the area in the entire environment that our car can observe.

Reward Lines. The reward line is 20-length-lines with location and reward distribution. They indicate rewards to the vehicle on its way to the end. They are distributed with equal distance along the track, and that distance can be manually tuned. We make the reward line sparse in the map by taking an interval of 15. It is then loaded into a planning list in the observation agent, and the current target is updated if the vehicle passes the line. The observer agent records the last reward line index passed and iterates over the following reward lines to find the line(s) vehicle passes. After that, the reward for passing is calculated based on the value of the grid of the line the car passed.

The agent checks if the vehicle has passed the current line on each frame. If it passes the line, the current reward line is updated to the next one based on the interval and saves the change into the history for observation. The history includes all the updates in the last four calls in each frame.

Vehicle State. The vehicle state includes the location, rotation, and velocity. There are three approaches to presenting the state. The first one uses the frame stacked

egocentric map to represent the velocity, using the map rotation to show the vehicle rotation, and maintaining the map centered around the vehicle. The second approach centers the map on the center of the reward line so that the vehicle’s movement also determines the rotation, and the relative location is calculated. Finally, the third approach is to input those features directly without rotating or translating the map. Our project implemented all three methods and chose the first one as the baseline.

Reward Line Generation. The reward line centers are along the planned trajectory and saved in a file. There are several trajectory designs, the center of the road, the minimum curvature line, and the car racing line. The project is currently using the centerline. After loading the centers, the direction of each reward line is calculated based on the direction normal to the direction from the last center to the current center. The centers are picked based on the set interval to have a different distance for different settings. The reward line is generated based on the location and direction with a length of 20. The value of the reward is calculated based on the distribution. We used a Gaussian distribution with a standard deviation of 10 to encourage the vehicle to go towards the center. After discovering that the car can learn the racing line in the long term, we switch the distribution back to uniform.

The reward lines are plotted on the map based on the vehicle’s relative location. Previously, the project used to only plot the following reward line. However, now, it plots all the reward lines within sight of the observation space to help the PPO agent with future planning.

Stacked OGM. Our simulation input data begins by grabbing a small portion of the complete environment map. In our case, the frame size is 84x84 pixels. Then we separate the perceived model input, a bird’s eye view of the environment-centered vehicle, and split them into individual channels. The Map Data channel contains the representation for collision borders. The Reward Data channel has the representation for the reward lines in front of the vehicle that has yet to be crossed. Finally, the Vehicle Data

channel contains the representation of the car. Together, these three channels represent the frame data of a given point in time. We combine the current frame data with the frame data of three previous points in time to make the complete input data stack to our CNN model, as shown in figure 1. It should be noted that all four data frames have an ego-centric orientation with respect to the vehicle of the current data frame.

This method of stacking four frames at different points in time is crucial to our model’s perception of change over time. The model can ‘see’ further and further into previous states in time by stacking additional frames and comparing the differences. This comparison leads to the perception of speed, change in direction over time, and even if the vehicle has some drift due to loss of traction.

Frame Skipping. Frame-skipping is a method we use to ignore specific frames of the environment. As you can see in the figure 4 below, we ignore three frames of input from the environment and observe the 4th. By doing so, we can tune how far back into the past our model can observe without slowing down the agent with larger input size. When the vehicle moves at slower speeds, this method can be very effective as there is very little change between any two given frames if the environment changes slowly. At faster speeds, this method becomes less necessary, and so it was not essential for the Full-Control model, which could achieve speeds of 195kph[**Ensure to update with final speed results**].

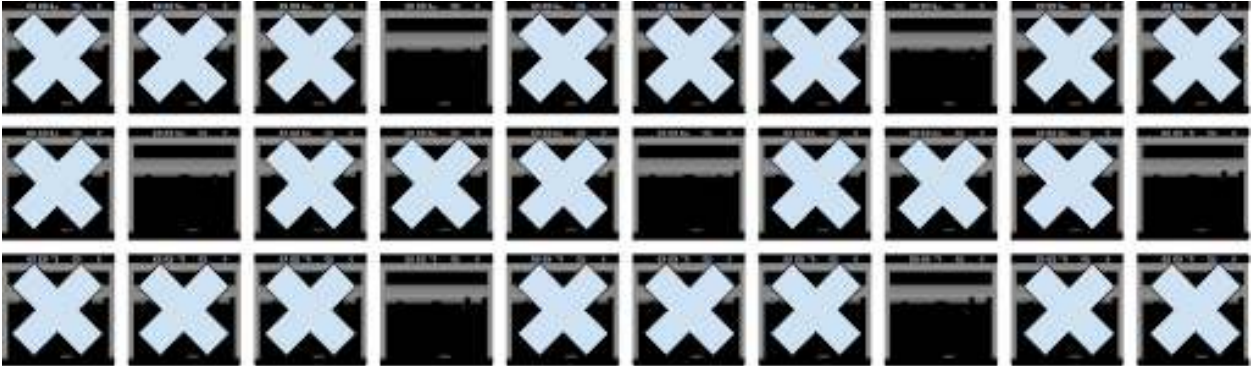


Figure 4
Frame-Skipping Visualization with 3 skipped frames

Feature Extraction Model

The feature extraction model receives the current state information from the observation agent and computes the pertinent features before feeding them into the Proximal Policy Optimization algorithm or PPO. The CNN model we use to do the feature extraction now is the same as the one in Atari since we believe that the model in Atari is powerful enough.

Proximal Policy Optimization Algorithm

The PPO On-Policy algorithm calculates action based on the input features and passes the action for the vehicle to the environment. The three agent outputs of PPO, throttle, steering, and braking, are in continuous action space. Throttle and braking are both bounded from $[0,1]$, and steering is bounded from $[-1,1]$. These outputs are then passed to CARLA to execute in one simulation frame.

Loss Function. PPO is a policy gradient method that introduces a probability ratio to its loss function which is then clipped. This keeps any one update from being too large or too small which greatly helps this method from skipping over a potential reward peak. (Daryl & Daniel, 2021)

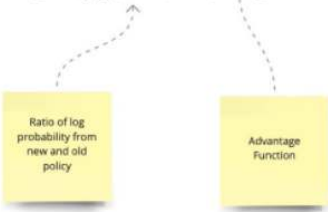
$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r_t(\theta) \hat{A}_t].$$


Figure 5

Loss Function with probability ratio

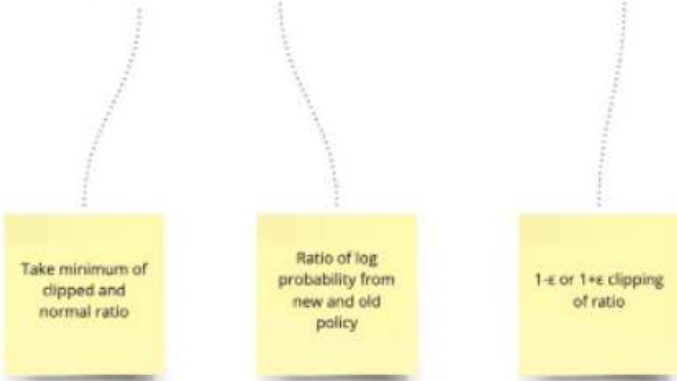
$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$$


Figure 6

Loss Function with probability ratio after clipping

Random Action. Each time the PPO agent generates a random action, it generates a value with an average of 0 and an std of 1. Thus, the action space is changed for throttle, steering, and braking for the first to have a high average value and the rest to have a low average value. In this way, the vehicle can have better results when driving randomly. As a result, the action space for the throttle is set from -2.5 to -0.5 so that it has a high probability of generating a high value. The action space for the steering is set to -5 to 5 so that the absolute value is small most of the time. The action space for the

braking is set to 1 to 3 so that the brake value is 0 for most of the time to speed up.

PPO Agent setting. The number of inputs per second in CARLA is set to 8. The beta, or entropy coefficient, value is set to 0. The number of time steps per rollout is set to be 3000, which is about 60s.

Calculation of Action and Reward

The environment gets the action from the PPO agent and uses it to control the vehicle in CARLA. After the state is updated in the CARLA and then updated for the observation agent, the reward to reward or punish a certain action is calculated to train it towards the desired behavior. The reward rewards on crossing the reward line to drive further. It also punishes crashing to avoid driving into the wall and being alive to move all the time at high speed.

Evaluation

Steering only control

Initial Baseline Run. Initial Baseline for a converging model that was capable of completing the Berkeley Minor Map. Our Reward function grants a reward for crossing a checkpoint, penalizes for crashing, and gives a very small penalty every step. We will refer to this last penalty as the "Hot Water" penalty and is what we will use to encourage to model to complete the course as fast as it can. It should be noted that the complete course is 5175 checkpoints.

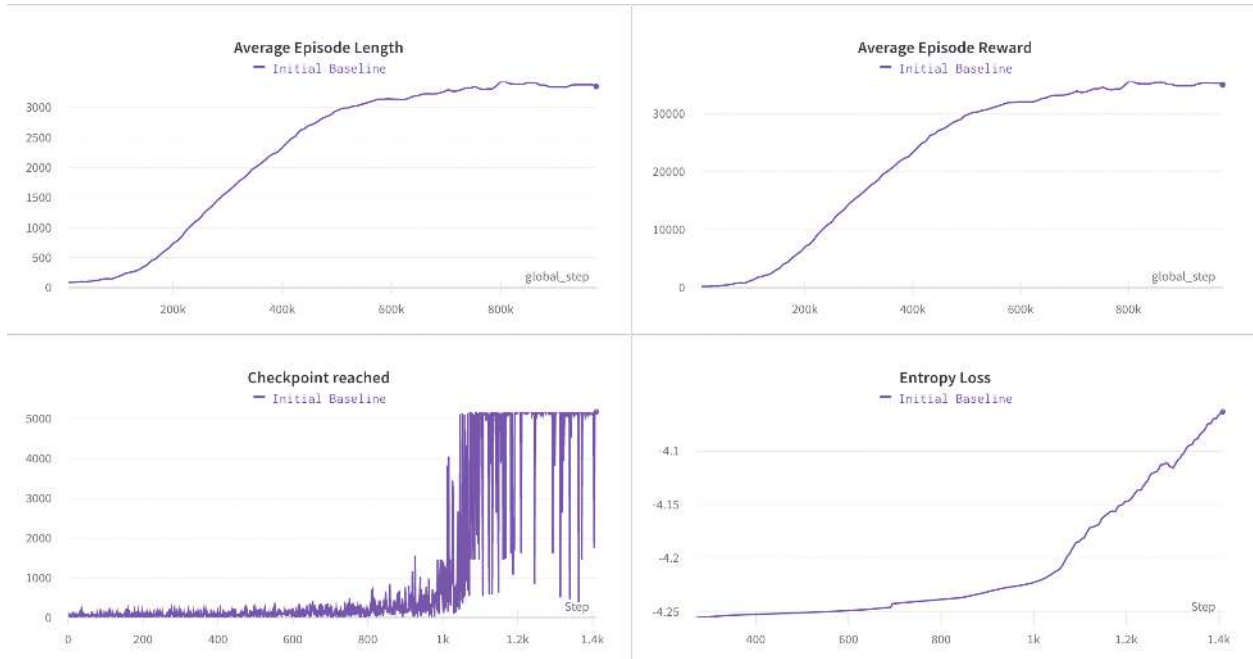


Figure 7

Initial Baseline Run Performance

Scaling Checkpoint Reward. In this series of tests, we introduce a scalar variable to adjust how much reward we grant for crossing a given line. For the following runs, our reward function is as follows:

- Reward for crossing a checkpoint = $\text{cross_location} * \text{checkpoint_interval} * \text{time_to_waypoint_ratio}$
- Penalty every frame ("Hot Water penalty") = -1
- Penalty for crashing = -200

where: cross_location = a value ranging from 0.5 to 1.0 based on where the vehicle crosses the checkpoint (see Figure below) $\text{checkpoint_interval} = 15$, this refers to the checkpoint density that we are working with $\text{time_to_waypoint_ratio}$ = a value used to scale the reward for each checkpoint

	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	
36	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
37	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
38	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
39	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
40	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
41	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
42	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
43	0.0000	31.79241	41.39319	52.41682	64.55779	77.32266	90.09766	102.09492	112.51836	120.60983	125.74141	127.50000	125.74141	120.60983	112.51836	102.09492	90.09766	77.32266	64.55779	52.41682	41.39319	0
44	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
45	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
46	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
47	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
48	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
49	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
50	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
51	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
52	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
53	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
54	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
55	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
56	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
57	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
58	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
59	0.0000	31.79241	41.39319	52.41682	64.55779	77.32266	90.09766	102.09492	112.51836	120.60983	125.74141	127.50000	125.74141	120.60983	112.51836	102.09492	90.09766	77.32266	64.55779	52.41682	41.39319	0
60	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
61	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
62	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
63	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
64	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
65	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
66	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
67	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0
68	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0

Figure 8

Gaussian Reward line example

In these runs, we tested different the values 0, 0.25, 1, 2, 4, and 8 for time_to_waypoint_ratio (Referred to below as 'R'). The Initial Baseline Run is also included for reference.

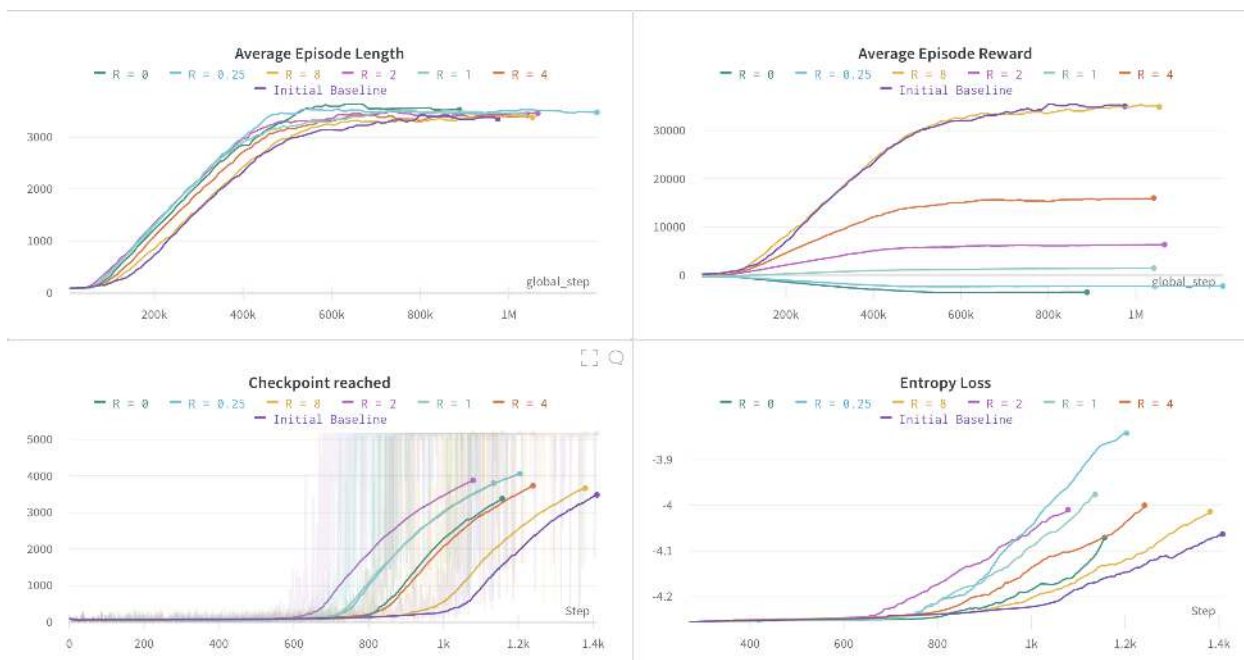


Figure 9

Performance results for different 'R' values compared with the Initial Baseline Run

We can see from the Average Episode Length that all runs converge to a similar curve. After 1 million global steps, the performance of each model is pretty similar. What we were interested in this test is how quickly the model learned to complete the course and how confident the model became over time which we observed in the Entropy Loss. We found that when the `time_to_waypoint_ratio` was set to 0.25 and 1, the model had a good balance of course completion time and confidence over time. What we found most interesting was the fact that the model continued to train and complete the map despite having a net negative reward when the `time_to_waypoint_ratio` was less than 1.

Negative Reward Limit. Upon observing a model that converged with a negative net reward, we determined that there should be a reward function where the model should optimize immediate termination after completing the entire course. The reward function we tested on this run is as follows:

- Reward for crossing a checkpoint =
 $\text{cross_location} * \text{checkpoint_interval} * \text{time_to_waypoint_ratio}$
- Penalty every frame ("Hot Water penalty") = -1
- Penalty for crashing = -25

where: `cross_location` = a value ranging from 0.5 to 1.0 based on where the vehicle crosses the checkpoint (see Figure below) `checkpoint_interval` = 15, this refers to the checkpoint density that we are working with `time_to_waypoint_ratio` = 0.25, a value used to scale the reward for each checkpoint In both runs below the net reward is negative as they both have a `time_to_waypoint_ratio` of 0.25. The only difference is that penalty for crashing is significantly less in this test.

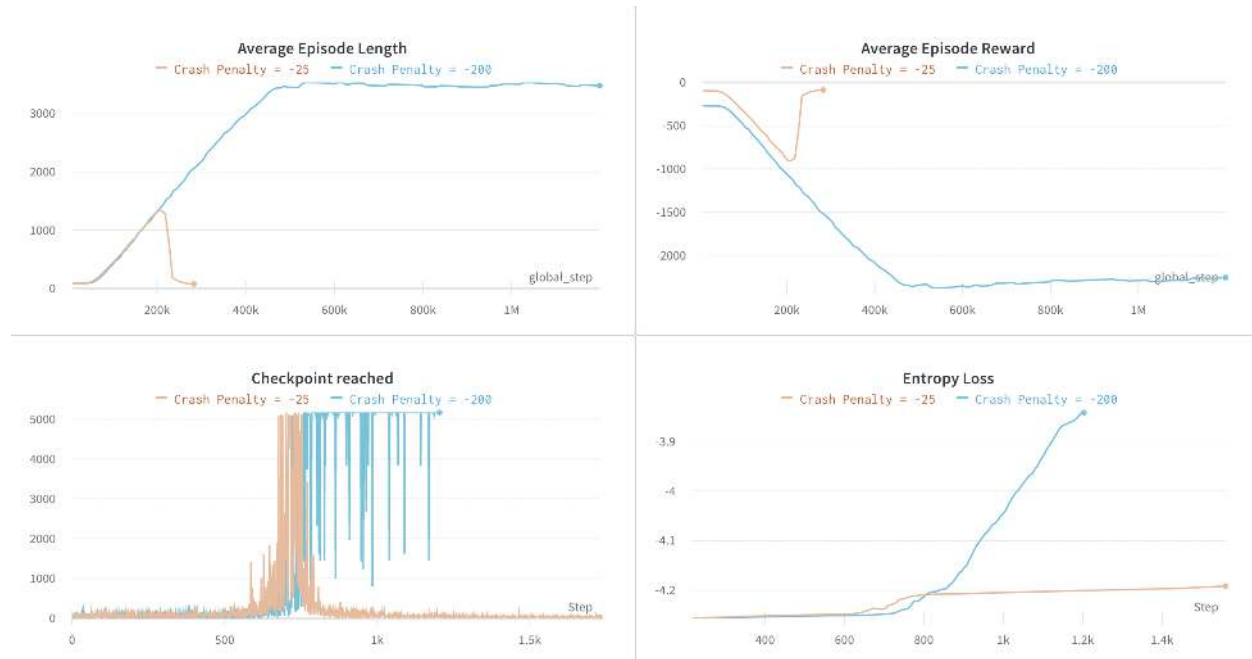


Figure 10

Performance results when exploring the Negative Reward Limit

This behavior shows that the model explored the entirety of the course and determined that it would not be able to achieve a better result than turning and crashing into the wall immediately. This also shows that despite course completion having a net reward of -2300, the model will continue to optimize the course until the net reward is greater than -200 or the crash penalty. We calculated the theoretical minimum number of frames (and therefore the minimum amount of "Hot Water" penalty) and determined that it is possible for the model to overcome the crash penalty given the amount of total available reward from checkpoints.

Flattened Reward Line. The following tests observe the results of making the reward for crossing a reward line even across the line such that the model receives the same reward no matter where it crosses.

$$\text{Reward for crossing a checkpoint} = \text{checkpoint_interval} * \text{time_to_waypoint_ratio}$$

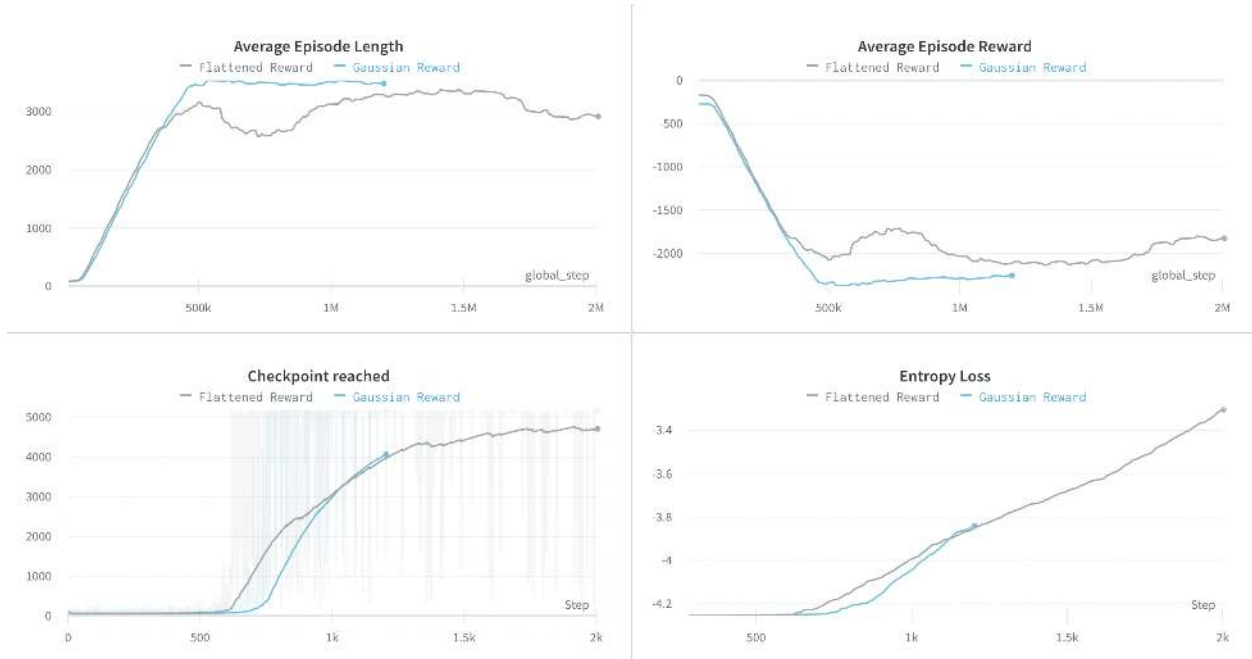


Figure 11

Performance for Flattened vs Gaussian Reward Lines, $time_to_waypoint_ratio = 0.25$

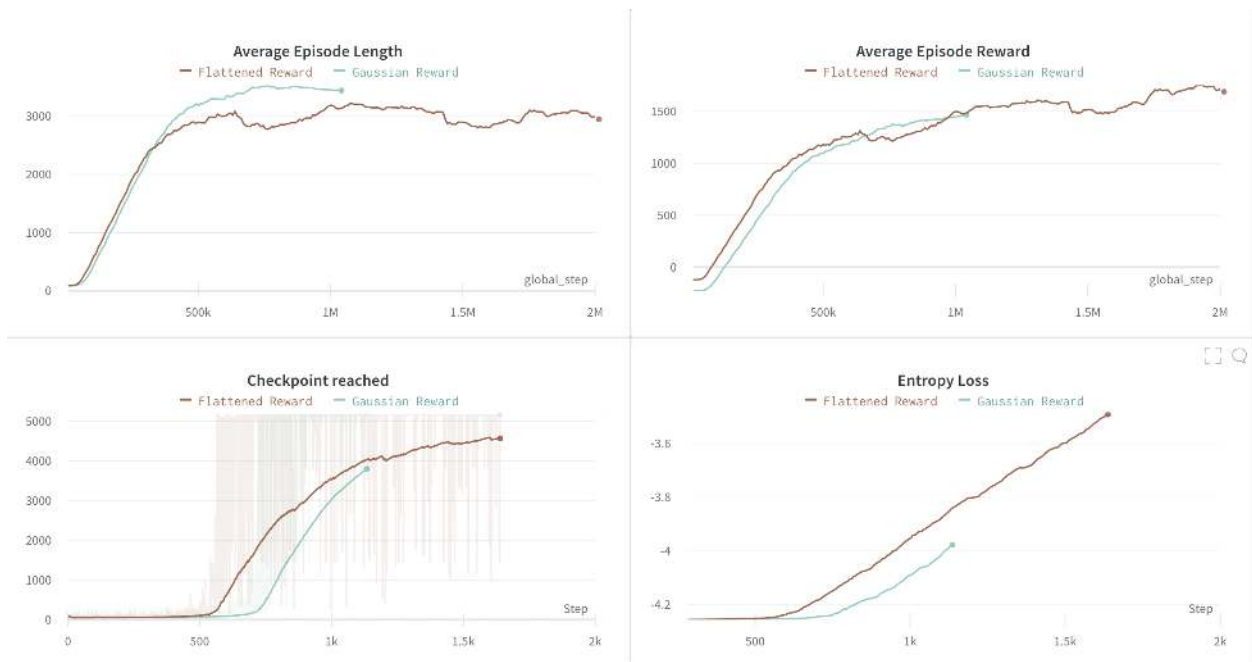


Figure 12

Performance for Flattened vs Gaussian Reward Lines, $time_to_waypoint_ratio = 1$

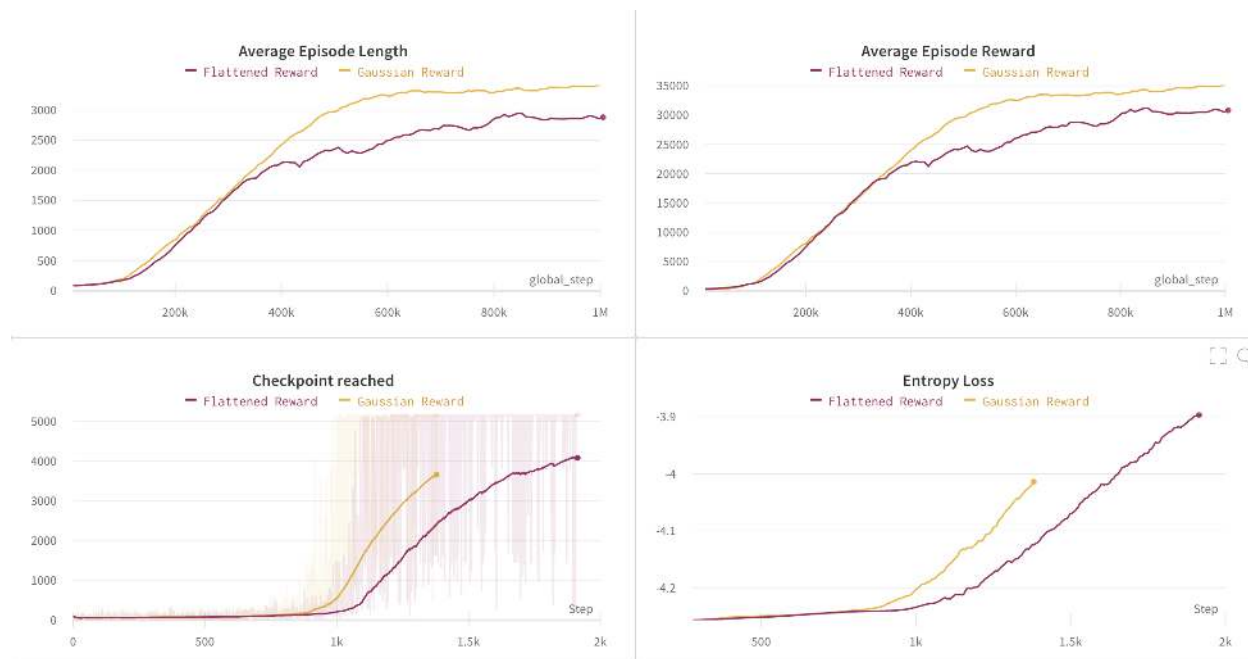


Figure 13

Performance for Flattened vs Gaussian Reward Lines, time_to_waypoint_ratio = 8

Highest speeds reached in a given run. In all runs the throttle is fixed to %80 of full throttle.



Figure 14

Speed Performance for Flattened vs Gaussian Reward Lines

These tests show that the flattened reward line can result in models that are not as consistent. However, in the final graph for Highest Speed Achieved, we can see that models are able to reach higher speeds more often with a flattened reward line. When the reward line is shaped by a Gaussian, there is an implicit racing line that the model is encouraged

to follow. While this may lead to more consistent results, this limits how the model explores the track. Without the implied racing line, the model has no guide and thus crashes into the wall more frequently while exploring the space. The benefit to this is that despite the increase in crashes, models with a flattened reward line are able to develop their own racing line naturally leading to faster times.

New Baseline Run. Moving forward with the most promising model, we ran an identical run to observe for any variance. We ran this new baseline for 6 million timesteps.

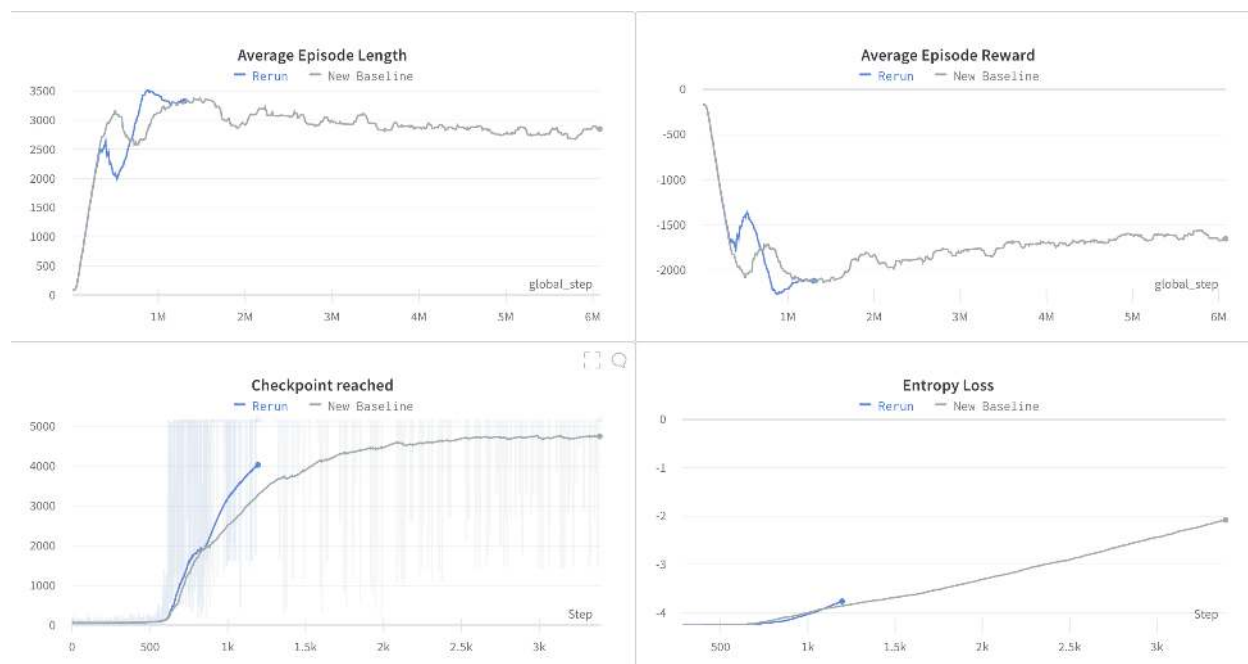


Figure 15

New Baseline performance results

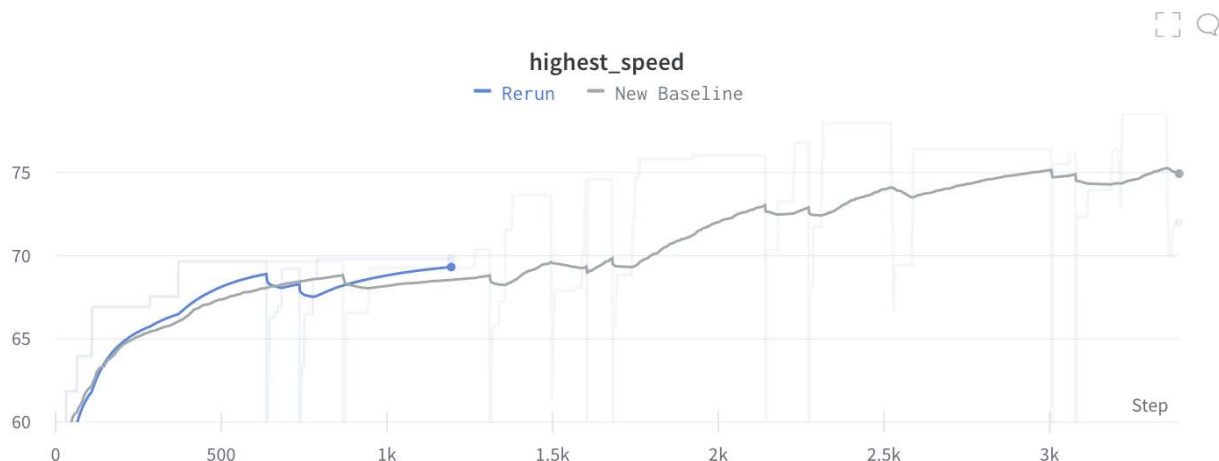


Figure 16

New Baseline top speed results

The New Baseline was run for several million timesteps and developed a very high track completion rate. While the steering only model has no control over the throttle, it is still able to reach higher and higher speeds by learning to reduce wobble. At the end of training, the model has developed a clear and efficient racing line.

Policy Gradient Tests. Next we ran several tests to adjust the value function coefficient. By doing so we can adjust the value loss and allow the model to focus more on the Policy Gradient as it was otherwise having very little effect on the model.

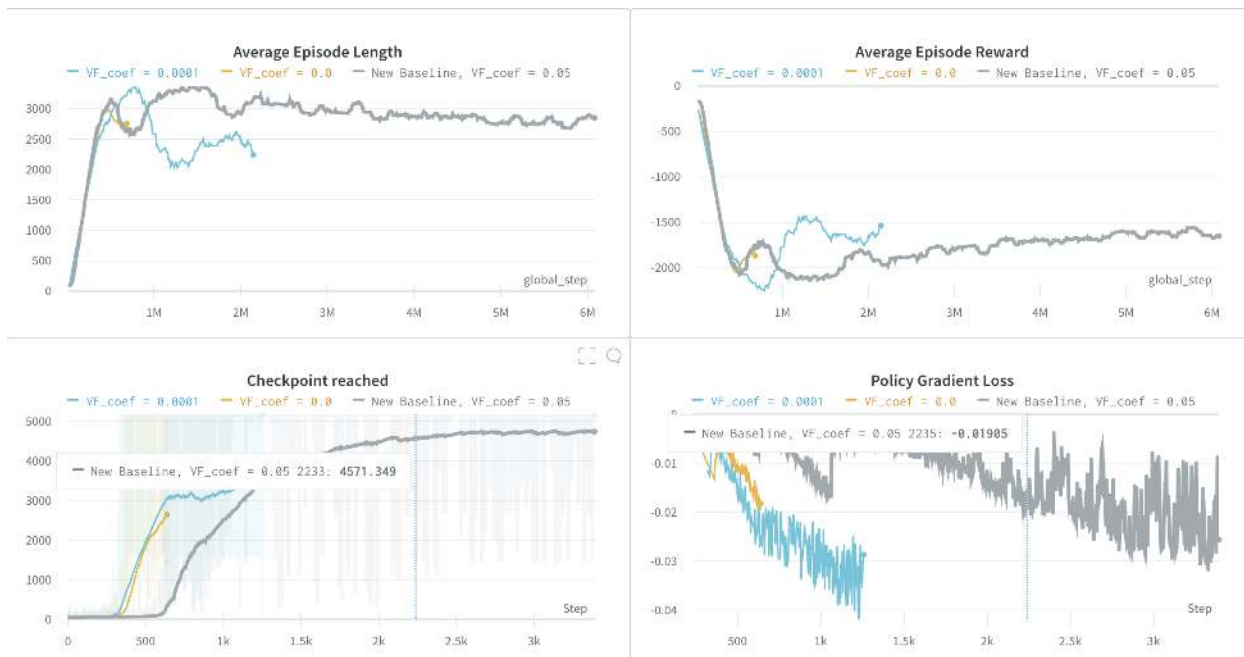


Figure 17

Policy Gradient Test results for $time_to_waypoint_ratio = 0.25$

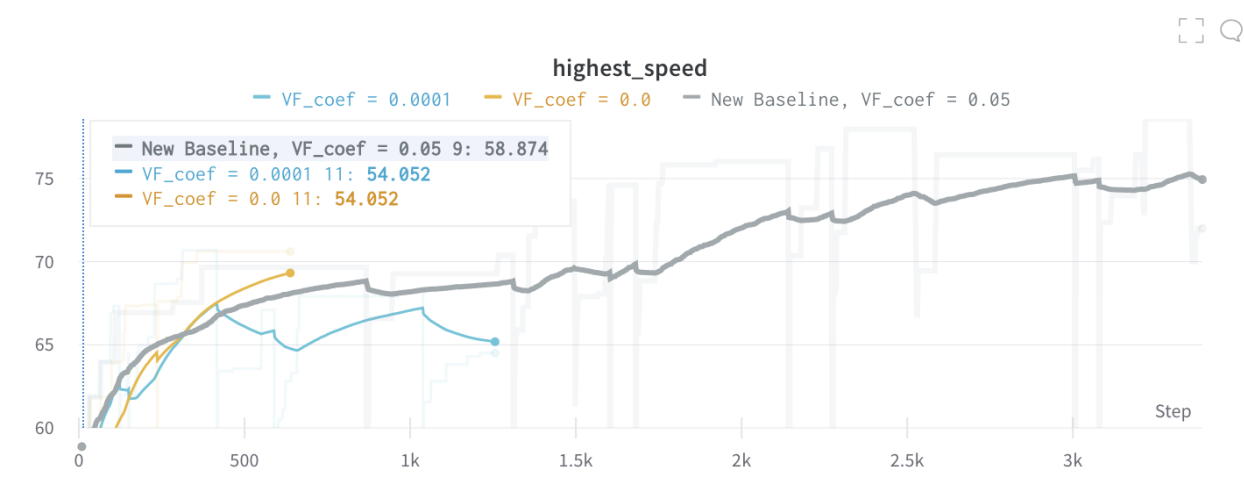


Figure 18

Policy Gradient Test top speed results for $time_to_waypoint_ratio = 0.25$

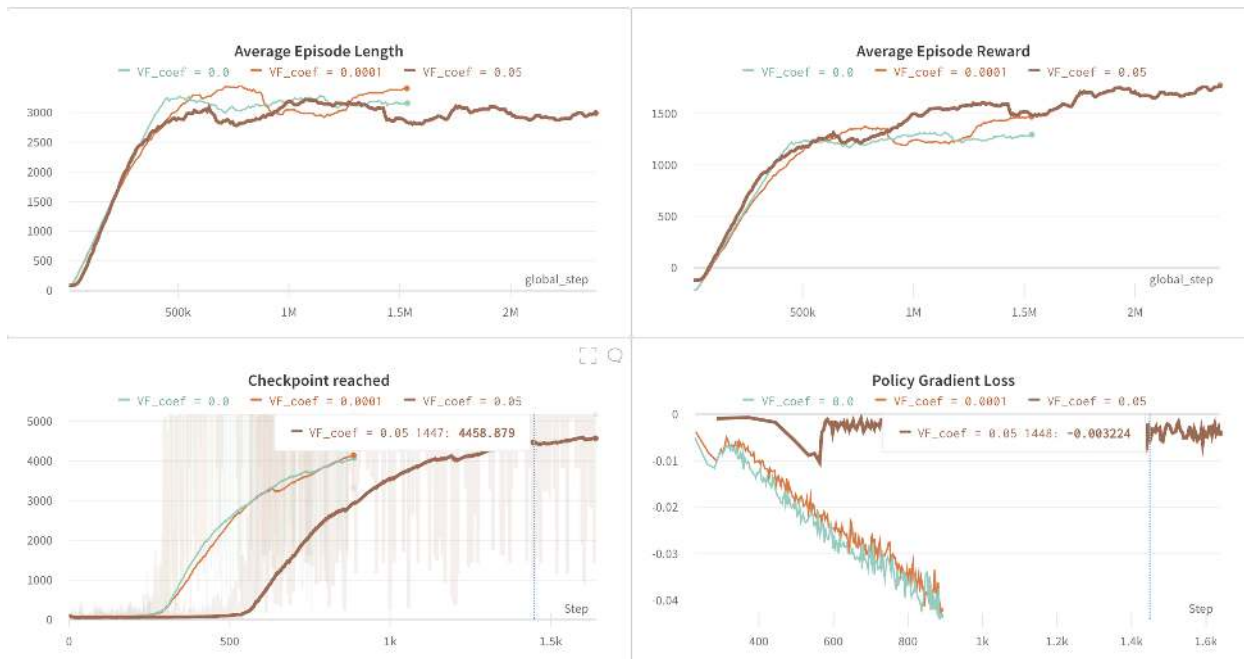


Figure 19

Policy Gradient Test results for time_to_waypoint_ratio = 1

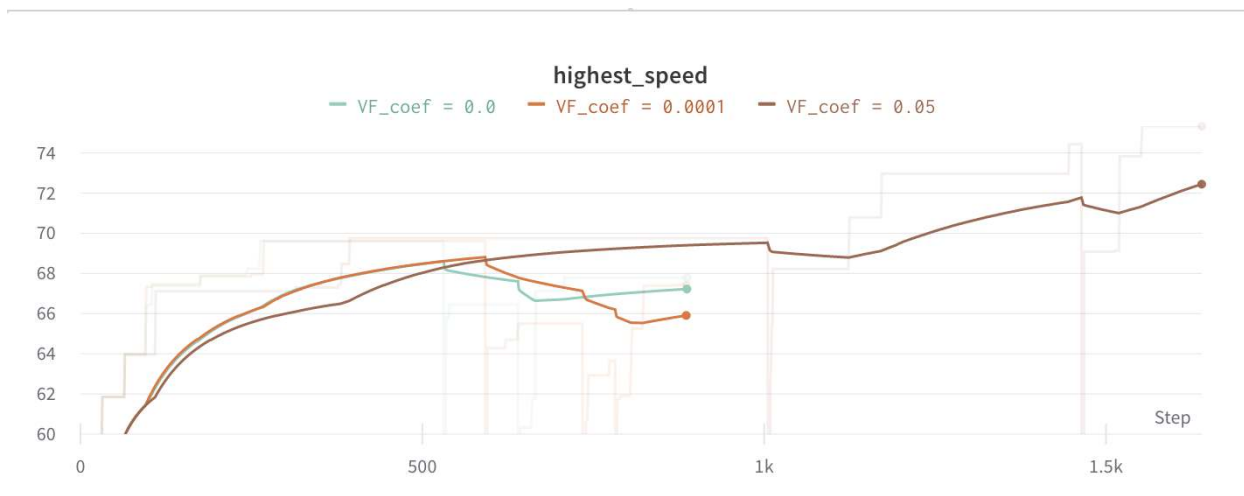


Figure 20

Policy Gradient Test top speed results for time_to_waypoint_ratio = 1

These tests show that by increasing the focus on the Policy Gradient, the model can learn to complete the course very quickly. However, we noticed that was due to the model heavily avoiding the crash penalty. When we observe the speeds achieved and the model's

performance on the course, we see that models with a larger emphasis on the Policy Gradient are "safer" drivers but not necessarily faster drivers.

Learning Rate Tests. These tests explore the effects that various Learning Rates have on our model when compared to the New Baseline where the Learning Rate is set to $1e-5$. This will adjust how quickly the model can change and adapt. The larger the Learning Rate(LR), the faster the model can change.

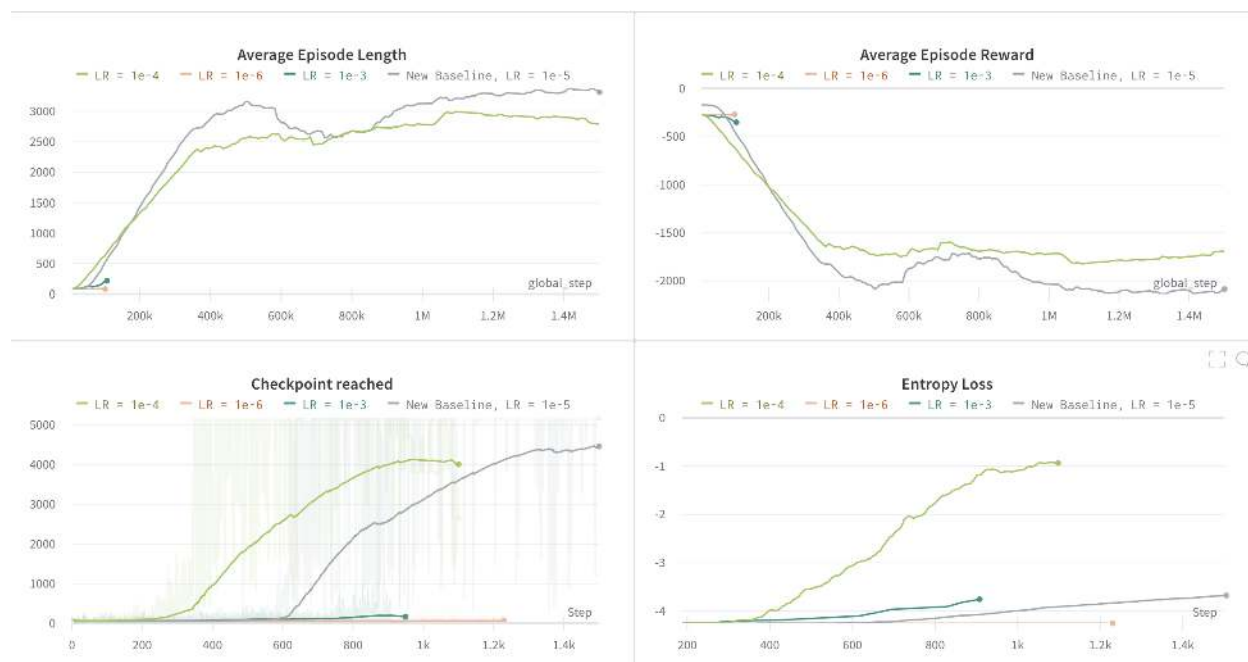


Figure 21

Learning Rate Test results for $time_to_waypoint_ratio = 0.25$

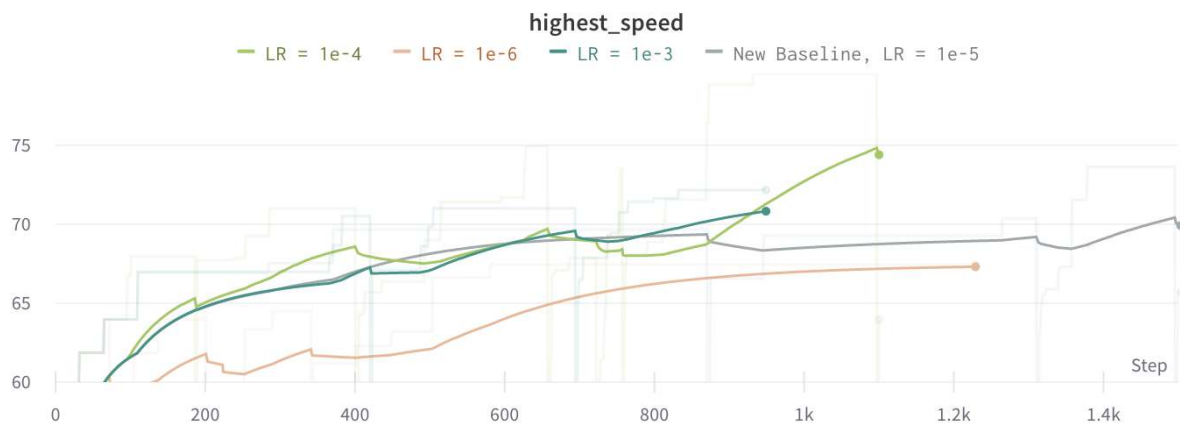


Figure 22

Learning Rate Test top speed results for $time_to_waypoint_ratio = 0.25$

These tests show some upper and lower bounds for learning rates that can be used in future runs. The slightly larger learning rate of $1e-4$ showed especially promising performance and should be considered in future runs.

Full Control

For Full Control, we adapted what we learned from the Steering Only series. While this model does have control of Throttle, Steering, and Braking, it is still very limited. Binary Full Control: While steering is calculated in the same way as before, Throttle and Braking are determined from a single action output instead of two. This action output is mapped from 0.0 to 1.0. If this action is greater than 0.5, throttle is set to 1.0 (The maximum throttle) and braking is set to 0.0. If the action is set to 0.5 or lower, the throttle is set to 0.0 and the brake is set to 0.8 (80% the maximum braking value).

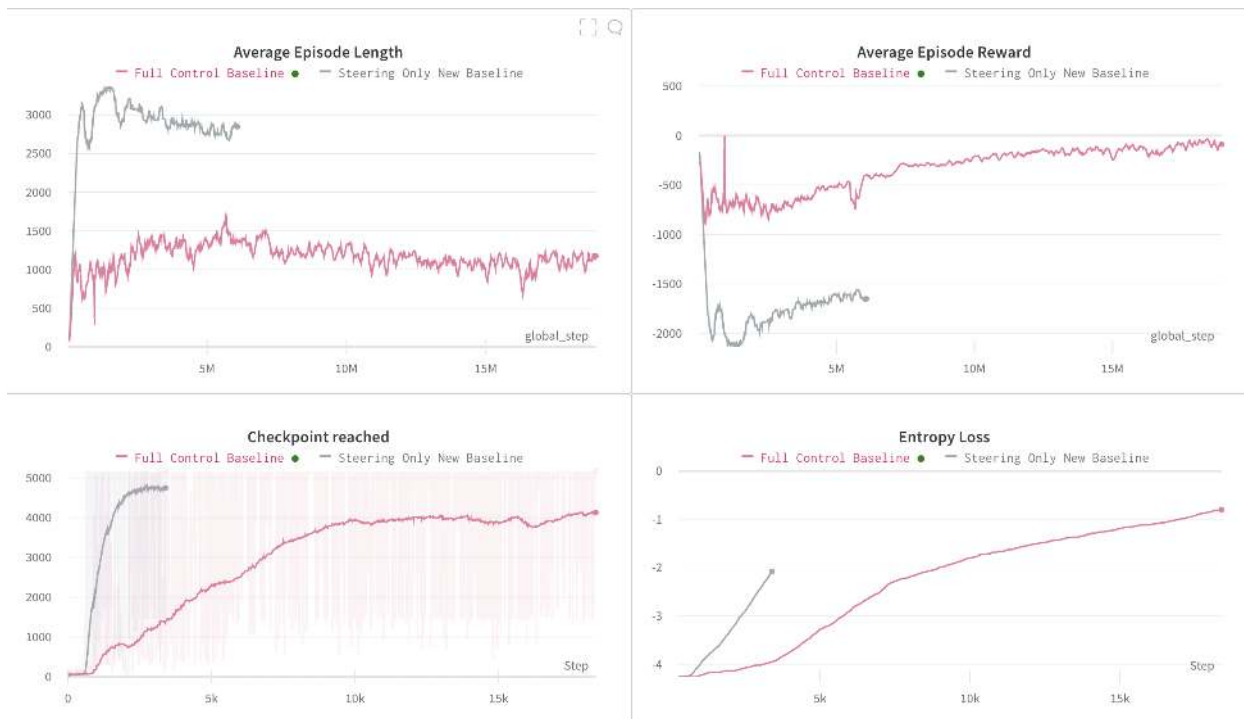


Figure 23

Full Control Performance results compared to Steering only New Baseline

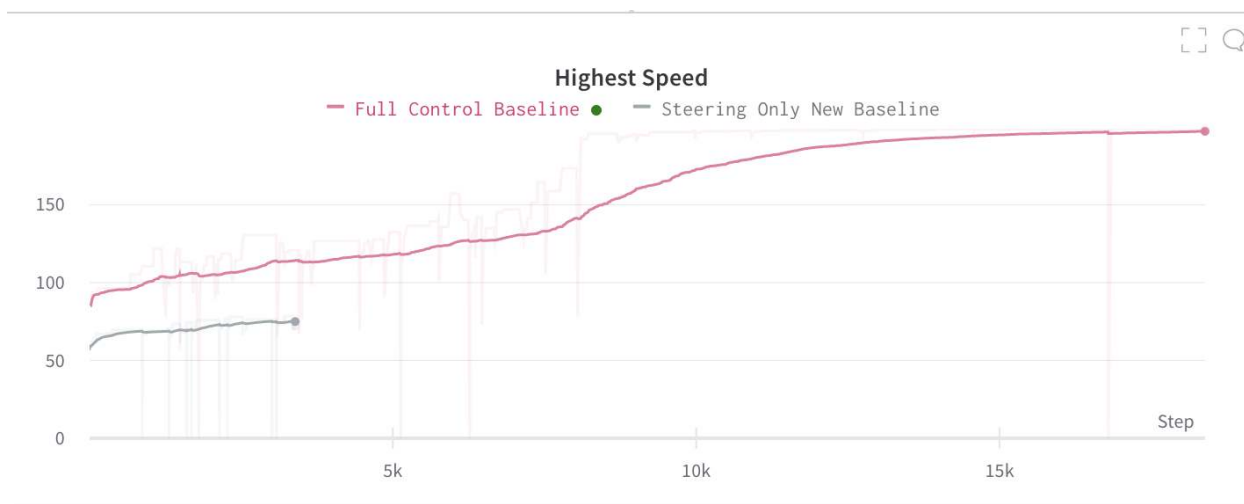


Figure 24

Full Control Performance top speed results compared to Steering only New Baseline

We trained this model for over 18 million timesteps and achieved very competitive results with a fastest time of 160.625 seconds (2 minutes and 40.625 seconds) for a single

lap when maximum vehicle speed is set to 200kph. While many of the things we learned from steering only were able to be immediately adapted to full control, there are plenty of new behaviors we had to adapt to during the course of this run. Stalling, turning around, wobble, and drifting: With the ability to brake, we noticed early on that the model would learn to simply sit idly and not make any progress. To combat this we added a stalling penalty that check if the vehicle was stalled for longer than 10 frames(since the vehicle begins from a standstill). Given its new action space, the model found it was able to turn completely around on the track and began to explore the space behind it. This was not a problem in steering only as the model was never able to do this with the fixed throttle. A fix was implemented that penalized the model the same way a crash did whenever reverse progress was made on the track. One thing we noticed the model was struggling to deal with was maintaining a straight path. This slight wobble was exaggerated the faster it went. We realized that this was due to how we set up the action space for Steering. While it is continuous from -1.0 to 1.0, it is virtually incapable of setting the steering value to 0. And so we implemented a "Deadzone" range that would set the steering value to 0 when the action was smaller than 0.001 and gave a very small reward to the model every time the "Deadzone" is triggered. While this did not eliminate the wobble entirely, we did see wobble reduction as well as higher top speeds. As the model began to achieve faster and faster speeds, the vehicle began to lose traction and drift. We made no change to the model and simply observed its behavior as it learned. Over time, the model began to apply drift turning to its solution on different corners. On the final turn which has the sharpest angle, the model has found plenty of success by applying a sharp drift to finish the race.

Berkeley Major

Our approach for the Berkeley Major map has been similar to the Berkeley Minor map approach in that we first attempted a steering only model. While it is able to complete the course, albeit with an extremely slow time of over 30 min, this method is

extremely consistent and prone to failure in several parts of the map. Certain sections of the track have divots that cause the vehicle to become stuck or crash the environment. Setting a low throttle value increases the chance of getting stuck, while a value that is too high will leave the model unable to make some of the tighter turns on the map. Due to the map size, the implementation of multiple spawn points on this map has greatly accelerated our training time. As "easier" sections of the map are consistently completed, we can shift the focus onto tougher sections that our model struggles with. In the graphs below, we can see the impact that shifting spawns has on the model's progress.

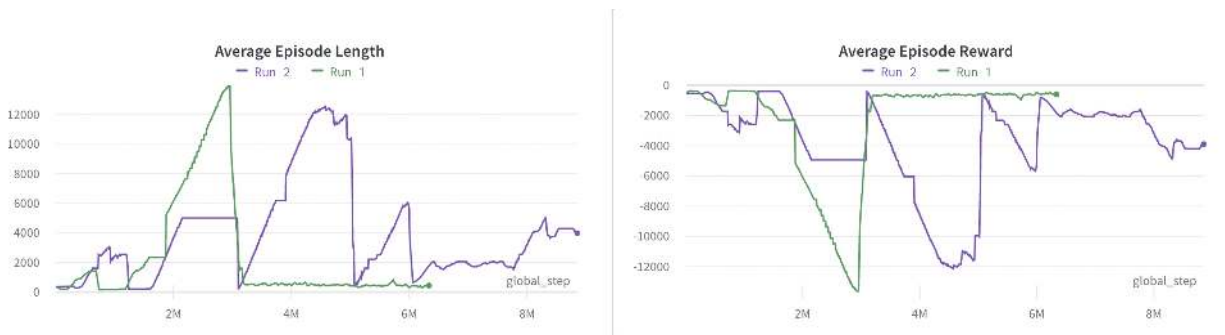


Figure 25

Berkeley Major Steering Only Performance results

Our plan moving forward is to apply our Full Control method to overcome these issues. Its ability to vary speed will be essential, along with making changes to the Occupancy Grid Map that will conform to the sections of the track that are problematic.

Conclusion

After training for about 8 hours, our model can converge and control the car to race to finish the lap. With the benefit of the simulator, the training process does not need any external label and would not cause any harm while crashing into the wall. As a result, the cost would also be cheaper than the self-driving learning process in the real world. This shows the power of our end-to-end solution, but we still have much work to do to achieve a competitive result in the car racing game. Our next step is to design a better feature

extraction model and apply our solution to the full control version of the car racing game, giving access to throttle and brake.

Future Work

Our model has shown that this method is more certainly viable and competitive against current PID-methods with far less user intervention. Still, there are certainly areas that can be improved upon or further researched. For starters, training time can be accelerated by integrating techniques such as Multi-Agent training where multiple vehicles are spawned at the same time and all experience is synthesized by the model. Other methods of improving training time include optimizing the existing framework in our repository for CARLA and the PC hardware.

Introducing obstacles for the model to avoid can create a challenge for the model that would be more true to the real-world scenarios a car runs into on a regular basis. Additionally, one can build upon this paper by implementing one of our trained models on hardware. This would come with the expected challenges of transitioning from a simulated environment to the real world, but would be instrumental in the effort to make autonomous vehicle research more accessible and affordable.

References

- Coad, J., Qiao, Z., & Dolan, J. M. (2020). Safe trajectory planning using reinforcement learning for self driving.
- Daryl, & Daniel. (2021). A graphic guide to implementing ppo for atari games.
- Kardell, S., & Kuosku, M. (2017). Autonomous vehicle control via deep reinforcement learning.
- Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., & Shah, A. (2018). Learning to drive in a day.
- Kuderer, M., Gulati, S., & Burgard, W. (2015). Learning driving styles for autonomous vehicles from demonstration. *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2641–2646. <https://doi.org/10.1109/ICRA.2015.7139555>
- Leurent, E. (2018). A survey of state-action representations for autonomous driving.
- Li, C., & Czarnecki, K. (2019). Urban driving with multi-objective deep reinforcement learning.
- Mania, H., Guy, A., & Recht, B. (2018). Simple random search provides a competitive approach to reinforcement learning.
- NHTSA. (2021). Automated vehicles for safety.
- Pomerleau, D. A. (1989). Alvin: An autonomous land vehicle in a neural network. In D. Touretzky (Ed.), *Advances in neural information processing systems*. Morgan-Kaufmann. <https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf>
- Sharifzadeh, S., Chiotellis, I., Triebel, R., & Cremers, D. (2017). Learning to drive using inverse reinforcement learning and deep q-networks.
- Wahlström, N., Schön, T. B., & Deisenroth, M. P. (2014). Learning deep dynamical models from image pixels.